Lesson 8: Understanding Interop

Sonja Keserovic
Program Manager

CLR Team

June 2004

Internal Technical Education

Understanding Interop

- After successfully completing this lesson, you will be able to:
  - Design a library that can be exposed to native clients easily (part #1)
  - Design a library that correctly utilizes native components (part #2)
- You will be able to understand the WHY behind your design but not necessarily HOW to implement it
- Multiple resources are available to address the HOW part

Part #1: Exposing Managed API to Native Clients

- Some examples
  - Microsoft® Office needs to use WinFS APIs
  - Shell needs to use "Avalon" APIs

Exposing Managed APIs
Exposing Options

- Expose COM-friendly APIs through common language runtime (CLR) interop
- Recompile native C++ clients with new C++ compiler as mixed-mode images (preferred)

- Expose static entry points from managed C++

Exposing COM-Friendly APIs
CLR Interop

- Every managed object is exposable as a COM object automatically…
  - Implements IUnknown, IDispatch, etc.
  - Can be registered (even side by side)
  - CLR interop takes care of ref counting, apartment transitions, etc.
- …But is usually not usable as a COM object automatically
  - Statics, parameterized constructors, generics are not COM-exposable
  - Versioning rules are very different

Exposing COM-Friendly APIs
FX Example for COM Usability/Versioning

- System.Math
  - Contains only static methods
- System.Collections.BitArray
  - No default constructor
- System.Type was AutoDual in v1.0
  - _Type interface was added in order to preserve backward compatibility

Exposing COM-Friendly APIs
Best Practices

- Design explicitly for COM clients
  - Separate managed-clients-only APIs and COM-friendly APIs
  - Make managed-clients-only APIs COM-invisible
  - Build COM-friendly APIs on top of managed-clients-only APIs (in a separate namespace)
  - For COM-friendly APIs: Implement interfaces explicitly, add new interfaces when making COM-breaking changes, etc. *(read docs for more rules)*

Exposing COM-Friendly APIs
Best Practices Example

```
namespace Sample.SerializeCD.Interop
{
    [ComVisible(true)]
    [Guid("[illegible]")]
    public interface ISerializeCD
    {
        //method definitions
        ...
    }
    [ComVisible(true)]
    [Guid("[illegible]")]
    [ClassInterface(ClassInterfaceType.None)]
    [ComDefaultInterface(typeof(ISerializeCD))]
    public class ComFriendlySerializeCD : ISerializeCD
    {
        //interface method implementation
        ...
        //private member to managed only API
        private SerializeCD zCD;
    }
}
```

Exposing COM-Friendly APIs
Pros and Cons

- Pros
  - Managed APIs consumable by variety of COM clients (Microsoft® Visual Basic® 6.0, any native C++ flavor)
  - All COM plumbing is taken care of by CLR interop layer
- Cons
  - Generating and maintaining two sets of APIs

Recompiling Native C++ Clients
C++ Interop (a.k.a IJW)

- Native C++ clients can be recompiled into IL code using /clr switch (mixed-mode image)
  - It's possible to recompile only parts that need to access managed APIs
- Recompiled clients have direct access to managed APIs
  - Can hold pointers to managed classes from native classes (using gcroot)

Recompiling Native C++ Clients
Best Practices

- Check latest guidelines and resources
- Analyze native client and decide which parts will need to be recompiled
- Check C++ documentation and other resources
  - Be aware of AppDomain transition issues
- Contact CLR interop team if you are not sure

Recompiling Native C++ Clients
Pros and Cons

- Pros
  - No changes required in managed library design
  - Every aspect of managed library accessible from recompiled clients directly
- Cons
  - C++ interop is not "magical"; it requires deep understanding of managed and native worlds

Exposing Managed APIs
Summary

- Consider C++ interop and client recompilation first
  - Make sure your clients are aware of C++ interop requirements
- Use CLR interop when recompilation is not an option
  - Make sure to follow best practices

Exercise: Case Study in Interop

- If you were WinFS, what kind of interop support would you offer to the Office team?
  - How would you make your decision?
  - What would you research first?

Part #2: Using Existing Native APIs from Managed Library

- Some examples
  - Shell needs to use native components (can't rewrite everything)
  - Microsoft .NET Framework uses native operating system APIs internally

Using Existing Native APIs
Wrapping Native APIs

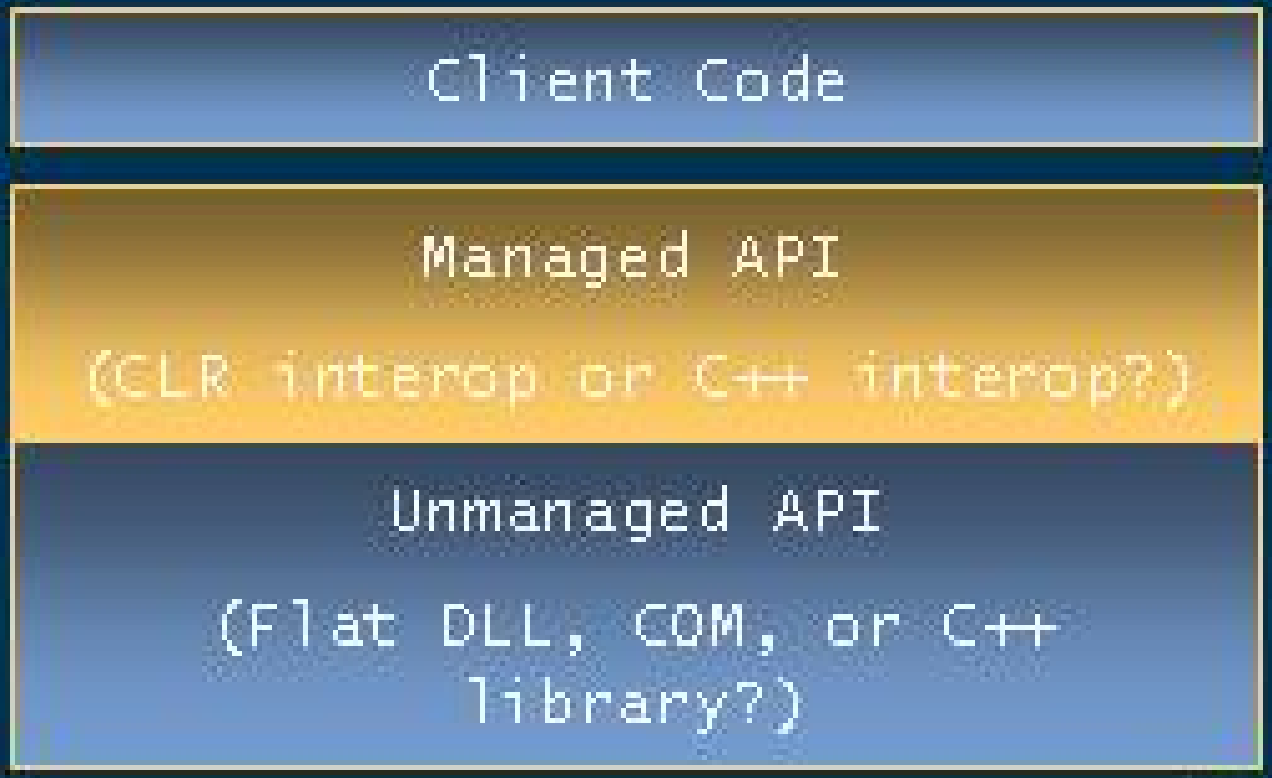
- Microsoft .NET is not a "rewrite everything" strategy
  - Interop is okay in most cases
- Owners of native APIs may choose to
  - Create a completely new set of managed APIs, that use old native APIs internally or
  - Expose a simple managed wrapper around their APIs (preferably while working on the above for the next version)
    - Publish primary interop assembly (PIA) for COM APIs (Microsoft Office XP and Microsoft Office 2003) or
    - Publish pinvoke definitions for flat APIs

Using Existing Native APIs
Types of Native APIs

- DLLs with static entry points (example: Microsoft Win32® APIs)
  - Use p/invoke (CLR interop); C# and any other language
  - Use C++ interop
- COM/Microsoft ActiveX® components (example: ADO)
  - Use COM interop (CLR interop); C# and any other language
  - Use C++ interop
- C++ library (example: Microsoft Foundation Classes [MFC])
  - Use C++ interop

Using Existing Native APIs
How to Choose Interop Technology?

Choose one that is best for your scenario ☺



CLR Interop
Pros and Cons

- Pros
  - Lots of things are done automatically: data marshaling, object lifetime, apartment/context transitions, exception and error handling
  - Plus you still can: fine-tune declarations for type safety and perf improvements, manipulate unmanaged memory directly, and be verifiable
- Cons
  - Writing p/invoke declarations for methods and structures in Visual Basic .NET or C# can be difficult
  - TlbImp.exe might produce incorrect interop assemblies if COM API is Interface Definition Language (IDL)–based
  - No compile-time check if native API is changing

C++ Interop
Pros and Cons

- Pros
  - Just include native header file, no wrappers or re-declarations needed
  - Compile-time checks
  - Full control over every aspect of interop transition
- Cons
  - No magic; requires deep understanding of both native and managed worlds
  - Handling AppDomain transitions is tricky today, especially for library development

Using Existing Native APIs
Best Practices for Using Flat APIs

- Both options work for all types of methods
- Use CLR interop (p/invoke)
  - Simple/easy to re-declare flat APIs
  - Limited number of flat APIs
- Use C++ interop
  - C++ is your language of choice
  - Very complex methods, with variable length structure parameters, for example
  - Compile-time check crucial because native code is changing too

Using Existing Native APIs
Best Practices for Using COM APIs

- Both options work for all types of COM objects
- Use CLR interop (COM interop) for:
  - Automation-compatible COM APIs
- Use C++ interop if:
  - C++ is your language of choice
  - COM API is IDL-based
  - You are fully aware of C++ interop requirements

Using Existing Native APIs
Best Practices for C++ Interop

- Check latest guidelines and resources
- Must use new MC++ interop templates: com_handle, com_handle_disposable, etc. because:
  - It's extremely hard to get all COM threading/lifetime/identity rules right from MC++
  - Templates promote correct usage patterns
- Make sure you understand rules for making AppDomain transitions
  - Guidelines will be available soon

Whidbey

Using Existing Native APIs
Best Practices for Using Native Handles

- Use SafeHandle or CriticalHandle for wrapping precious unmanaged resources like operating system handles

Whidbey

- Don't use SafeHandle and CriticalHandle if your resource is not subject to handle recycling attacks and you do not require critical finalization for it
  - They might add significant performance overhead
- They provide protection for handle recycling security attacks, critical finalization, and special managed/unmanaged interop marshaling
  - More information in the spec: http://devdiv/SpecTool/Documents/Whidbey/CLR/CurrentSpecs/Interop%20-%20SafeHandle%20Spec.doc

Using Existing Native APIs
Summary

- Pick appropriate technology for your scenario
- Check latest guidelines and resources
- Make sure that the native API you want to use isn't available as managed API already

Exercise: Case Studies in Interop

- If you were the "Avalon" team and you needed to use some very complicated native flat API, what would you do?
  - "Avalon" is written in C#

Using Interop in Both Directions
Best Practices for Interop Performance

- Know performance goals and measure against them often
- Make call worth the cost of the transition
- Fixed cost of a transition is relatively low
  - p/invoke: ~10 instructions; COM interop: ~70 instructions
- Total cost depends on marshaling
  - Primitive types and arrays of primitive types are almost free
  - Types that require transformations can be very expensive (Unicode<->ANSI)

Using Interop in Both Directions
Best Practices for Portability

- Make sure native parts of your application are available for 64-bit platform
- Do not use structures with explicit layouts except for unions (all field offsets are 0)
- Do not use int to represent pointers, use IntPtr instead

Using Interop in Both Directions
FX Example for Portability

IEnumVARIANT native definition

```
HRESULT Next(unsigned long celt, VARIANT
  FAR *rgVar, unsigned long FAR
  *pCeltFetched);
```

UCOMIEnumVARIANT definition

```
Int32 Next (Int32 celt, Int32 rgVar,
  Int32 pCeltFetched);
```

IEnumVARIANT Whidbey definition

```
Int32 Next (Int32 celt, Object[] rgVar,
  IntPtr pCeltFetched);
```

Best Practices Review
Library Design

- Know your options
  - Do not rewrite for the sake of rewriting
  - Use CLR interop or C++ interop when appropriate
- Be careful about performance
  - Know your goals and measure often
  - Make native/managed split wisely
- Use interop resources to get more information

Best Practices Review
Library Development

- Turn on Managed Debugging Assistants (MDAs, a.k.a CLR debug probes) during debugging
  - Enable you to "peek" into what's going on inside the runtime
  - Most MDAs are interop-related
- Enable mixed-mode debugging
- Run FxCop

Lesson 8 Summary

- CLR interop and C++ interop provide support for all scenarios where it's necessary to use existing native code
- No need to rewrite your existing applications/libraries
- Multiple interop strategies are available; choose the best for your situation
- Multiple interop resources are available; check them before making important decisions

Interop Resources

- Interop white paper and guidelines (Microsoft internal): http://msdn.microsoft.com/netframework/default.aspx?pull=/library/en-us/dndotnet/html/manunmancode.asp
- Book—.NET and COM: The Complete Interoperability Guide by Adam Nathan
- MDAs and CLRSPY tool: http://blogs.gotdotnet.com/anathan/PermaLink.aspx/976d1677-a6d1-4a25-9c7b-84488824b8267
- Chris Brumme's blog: http://blogs.msdn.com/cbrumme/
- Adam Nathan's blog: http://blogs.gotdotnet.com/anathan/
- CLR interop team Web site: http://team/sites/clrmst/default.aspx
- Contacts: "clriopds" alias, ANathan, DMortens, ChrisEck, and SonjaKe